Test-case Generation for Maude Specifications*

Adrián Riesco

Technical Report SIC-10/11

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

November 2011

*Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

Abstract

Testing is one of the most important and most time-consuming tasks in the software developing process and thus techniques and systems to automatically generate and check test cases have become crucial. In previous work we have presented techniques to test membership equational logic specifications that consist of two steps: first several ground terms are generated by using all the available constructor symbols in a breadth-first search, and then these terms are processed to check whether they fulfill some properties. This approach presents the drawback of separating two related processes, thus examining several terms that are indistinguishable from the point of view of testing. We present here a narrowing-based test-case generator that improves the performance of the tool and extends its use to rewriting logic specifications. First, we present two mechanisms to improve the narrowing commands currently available in Maude to use conditional statements and equational modules. Then, we show how to use these new narrowing commands to perform three different approaches to testing for any Maude specification: code coverage, property-based testing, and conformance testing. Finally, we present trusting mechanisms to improve the performance of the tool. We illustrate the tool by means of an example.

Keywords: testing, Maude, narrowing, coverage, property, conformance

Contents

1 Introduction

Testing is a technique for checking the correctness of programs by means of executing several inputs and studying the obtained results. Testing is one of the most important stages of the software-development process, but it also is a very time-consuming and tedious task, and for this reason several efforts have been devoted to automate [16, 2, 1]. Basically, we can distinguish two different approaches to testing: glass-box testing [13, 23], that uses the specific statements of the system to generate the most appropriate test cases, and black-box testing [31, 14, 5], that considers the system as a black box with an unknown structure and where a specification of the system is used to generate the test cases and check their correctness. We can also distinguish different kinds of testing depending on how the test cases are obtained: they can either be ground terms that are later executed to check the obtained results or terms with variables that are symbolically executed [19] to find the most appropriates values to test the program. While the former generates in general more test cases (because it just combines constructors to build terms) they can be illegal (input that can never be used in real executions) and equivalent (different test cases check the same statements), the latter generates less but more accurate test cases.

Maude [8] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic [21], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude functional modules correspond to specifications in membership equational logic [3], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude system modules are used to define specifications in this logic. The current version of Maude supports a limited version of narrowing [30], a generalization of term rewriting that allows to execute terms with variables by replacing pattern matching by unification, for some *unconditional rewriting logic theories without memberships*. This limitation is dropped in this work by using a program transformation and by checking separately the conditions.

As part of an ongoing project to test and debug Maude specifications, we have implemented a declarative debugger for Maude specifications [27], that allows the user to debug both wrong (incorrect results obtained from a valid input) and missing (incomplete results obtained from a valid input) answers in any Maude specification, and a test case generator for functional modules [26]. The testing approach used in that paper consists of different phases: first, the module is preprocessed to obtain the statements used by the functions being tested; then, terms are generated by using a breadth-first search that takes into account the constructor information provided by the user, and then each of these terms is executed step-by-step to check the used statements. However, this approach uses ground terms and, as explained above, presents an important drawback: since the test cases are not generated following the structure of the program but just the available constructors, most of them apply the same statements, hence consuming most of the time and preventing more complex terms from being checked due to the time and space constraints. This problem is solved here by symbolically executing terms with variables with narrowing, which tries to symbolically apply all the possible statements by unifying the appropriate variables, thus generating terms that traverse all the possible paths.

A simple example of the difference between the two techniques is illustrated with Java-like syntax in Figure 1. In this example, the function `min` computes the minimum between three natural numbers, and we want to test it by finding test cases executing all the instructions (the `return` ones) and checking the results. The generate-and-check approach would generate different tuples `(x, y, z)` by using the constructors (that we assume are `0` and the successor function for natural numbers, denoted by `s`), obtaining the values `(0, 0, 0)`, `(0, 0, 1)`, `(0, 1, 0)`, ... Note that these three tuples only use the `return x` instruction and that, in general, it will be the one executed by all the tuples where `x` is smaller than or equal to the rest of the values. This kind of equivalences forces the tool to generate several inputs to cover four instructions, making the whole process inefficient [25]. On the other hand, narrowing does not use specific values but variables, which are modified when required. For example, when the first condition is found narrowing would fix[1] `x` to `0` to make the execution to take the `then` branch and `x` to `s(x')` (with `x'` a fresh variable) and `y` to `0` to take the `else` branch; with these values the `return x` and `return y` instructions are executed. A second stage of the narrowing process would fix `x` to `s(0)` and `y` to `s(y')` (with `y'` a fresh variable) to take the `then` branch (the values for the `else` branch are computed similarly), and then `z` will take the value `0`; that is, the test cases obtained by using narrowing are `(0, y, z)`, `(s(x'), 0, z)`, `(1, s(y'), 0)`, and `(1, 1, 0)` (where we write `s(0)` as `1`). Note that in this case we have directed the search, hence discarding several equivalent inputs that were required by the previous method; note also that the variables can be fixed with simple values (with `0` in our case) to obtain "standard" test cases that can be executed as usual. Note however that generating and checking test cases is very useful when the error can be found with small inputs, due to its exhaustive search, or when using some other generation strategies like Quickcheck or Easycheck, that will be

[1] Assuming a standard definition for the <= function.

```
nat min(nat x, nat y, nat z){
  if (x <= y) { if (x <= z) { return x; } else { return z; } }
  else { if (y <= z) { return y; } else { return z; } } }
```

Figure 1: Minimum function for three natural numbers

outlined in the next section.

We present in this paper a program transformation to test Maude functional modules by using narrowing, a strategy to use membership axioms and conditional statements in the narrowing process,[2] and the adaptation of three testing techniques to Maude: two white-box approaches (one selects a set of test cases whose correctness must be checked by the user, while the other one checks whether a property holds in the specification) and one black-box mechanism (conformance testing). In the first case, in addition to other criteria described in [26], we have adapted a new criterion to select the set of test cases to be checked by the user in system modules, which is based on modified condition decision coverage [18] and checks the negative information (the rules that are not applied). Finally, we enhance the performance of the tool by providing trusting techniques that prevent the system from taking into account some statements. The transformation, the extension of the narrowing process, and the testing strategies have been implemented in a Maude prototype by using its meta-level capabilities, that allow to manipulate Maude modules and statements as usual data. Moreover, it also provides support for some predefined modules and attributes, such as `owise`, that indicates that the current equation is only used if no other cannot be applied.

The rest of the paper is organized as follows: Section 2 presents some related work and its relation with our system. Section 3 introduces Maude and narrowing, Section 4 describes a module transformation that allows us to use narrowing on Maude functional modules, while Section 5 presents how to use conditional rules in the narrowing process. Section 6 illustrates how the techniques described in the previous sections are used to generate test cases, while Section 7 presents some trusting techniques to improve the performance of the system. Section 8 sketches the implementation of the tool and Section 9 concludes and outlines some future work. The source code of the tool, examples, related papers, and much more information is available at `http://maude.sip.ucm.es/testing/`.

2 Related work

Different approaches to testing for declarative languages have been proposed in the literature. As explained in the introduction, test cases can be checked in different ways: executing ground test cases or symbolically executing terms with variables.

The first approach is followed by Smallcheck [29], a property-driven Haskell tool that considers that most of the errors can be found by using only a few constructors, and thus it generates all the possible combinations of constructors given a (usually small) bound on the size of the test cases. Another tool following this ground approach is Quickcheck [7], a test-case generator developed for Haskell specifications where the programmer writes assertions about logical properties that a function should fulfill; test cases are randomly generated by using the constructors of the data type (in contrast to the complete search performed by Smallcheck) to test and attempt to falsify these assertions. The project, started in 2000, has been extended to generate test cases for several languages such as Java, C++, Erlang, and several others. Finally, Easycheck [6] is a test-case generator for Curry that takes advantage of the non-determinism of funcional-logic programs to generate a tree of potential test cases, that is later traversed to list only the most interesting ones.

The second approach has been applied by Lazy Smallcheck [29] (an improvement of a previous system called SparseCheck), a library for Haskell to test partially-defined values that uses a mechanism similar to narrowing to test whether the system fulfills some requirements. Another way of achieving symbolic execution is by considering that the statements in the program under test introduce constraints on the variables, an approach followed by PET [15], that uses Constraint Logic Programming to generate test cases satisfying some coverages on object-oriented languages. Finally, narrowing has been used to verify security protocols [20, 17], symbolically exploring the state space trying to find a flow in the protocol.

The previous version of our approach is quite similar to Smallcheck: we generate the complete search space given the constructors and a bound, but we use them for both white-box and black-box testing, while Smallcheck only tries to disprove some properties. Note that, on the one hand, the strategies in our previous system could possibly be improved by following an approach similar to Easycheck, while on the other hand we can consider that

[2] This strategy allows all kinds of conditions: rewrite and equational conditions, solved by narrowing (the latter, which includes equational and matching conditions, requires a previous transformation), and membership conditions, solved by using unification.

the current narrowing approach is another way of pruning the tree of possible terms, making our approach similar to it. Regarding Quickcheck, it is an industrial tool with several heuristics and a lot of experience in testing, and hence it presents a better performance than our tool, that we try to improve by providing trusting mechanisms to the user. On the other hand, an advantage of our tool is the computation of test cases fulfilling different coverage criteria, which allows the user to test the specification by checking test cases "by hand" even when no properties over the specification are stated, and the usage of Maude as both a specification and implementation language, which allows to perform conformance testing using a previously tested Maude module as specification. Moreover, both Quickcheck and our tool implement the *shrinking* mechanism, that consists of returning the simplest form of a term that detects a bug in the program; in our case it is implemented by performing a breadth-first search using narrowing steps, that will find the simplest term (w.r.t. the number of steps) reproducing the buggy behavior. The more similar approach to ours is Lazy Smallcheck; both are narrowing-based experimental tools that focus on research rather than in efficiency, and thus they present a similar performance; however, Smallcheck only is applied to property-based testing. PET provides a coverage of the statements in Java-like programs, but it does not allow the user to state properties or check the correctness of the system against a specification. Finally, the verification of security protocols focus on a specific problem and cannot be compared with the rest of tools.

Note that, in general, each system only focus in one testing approach: coverage, properties, or conformance. Maude features allow us to implement a wide range of testing techniques: we can manipulate its modules to perform white-box testing by using its meta-level capabilities; its analysis tools (such as the `search` command) ease the testing of properties; and Maude programs can be used as specification of another ones.

3 Preliminaries

This section introduces Maude and its narrowing mechanisms [9].

3.1 Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [21] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [3], which, in addition to equations, allows one to state membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

Maude functional modules [8, Chap. 4], with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [8, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories [21]. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

An important characteristic of Maude functional modules is that sorts are grouped into equivalence classes called *kinds*; that is, all the sorts related by a subsort relation belong to the same kind [8]. Intuitively, terms with a kind but without a sort represent undefined or error elements. We will make extensive use of kinds to indicate that variables may have any sort when performing unification; the proper sorts of the variables will be later checked by means of membership axioms.

We introduce Maude modules with an example; variable declarations are not shown because of space constraints, but assume they are defined at the sort level. We specify ordered lists of natural numbers in the following module:

```
(fmod SORTED-NAT-LIST is
  pr NAT .
```

We use the sort `NatList`, with constructors `nil` and `_._`, for generic lists and `SortedList` for sorted lists, which are a subsort of `NatList`:

```
  sorts SortedList NatList .        subsorts SortedList < NatList .
  op nil : -> SortedList [ctor] .   op _._ : Nat NatList -> NatList [ctor] .
```

We use membership axioms to characterize nonempty sorted lists.[3] They indicate that the singleton list is ordered (`ol1`) and that a larger list is ordered if the first element is equal to or smaller than the second one and the rest of the list is also ordered (`ol2`):

[3]Note that the empty list is a sorted list because of the operator declaration.

```
  mb [ol1] : N . nil : SortedList .
  cmb [ol2] : N . N' . L : SortedList if N <= N' /\ N' . L : SortedList .
```

We also specify a function `ins-sort` that sorts a list by inserting the elements in an ordered fashion by using the auxiliary function `ins-list`:

```
  op ins-sort : NatList -> SortedList .
  eq [is1] : ins-sort(nil) = nil .
  eq [is2] : ins-sort(N . L) = ins-list(ins-sort(L), N) .
```

This function returns the singleton list when inserting an element into the empty list:

```
  op ins-list : SortedList Nat -> SortedList .
  eq [il1] : ins-list(nil, N) = N . nil .
  ceq [il2] : ins-list(N . SL, N') = N' . (N . SL) if N' <= N .
  ceq [il3] : ins-list(N . SL, N') = N . ins-list(SL, N') if N < N' .
```

For nonempty lists, if the element being inserted is smaller than the first one in the list, the new element is put as new first element:

```
  ceq [il2] : ins-list(N . SL, N') = N' . (N . SL)
   if N' <= N .
```

Otherwise, the first element of the list remains unchanged and the insertion continues with the rest of the list:

```
  ceq [il3] : ins-list(N . SL, N') = N . ins-list(SL, N')
   if N < N' .
endfm)
```

Since we are also interested on testing system modules, we use this module to specify how processes enter into a critical section in the following system module `CS`. We consider that processes are represented by their priority (the smaller the number the higher the priority), and hence lists of natural numbers stand for lists of processes:

```
(mod CS is
  pr SORTED-NAT-LIST .
```

We define the sort `NatSoup` for an associative and commutative multiset built with the operators `mtSoup` and `_,_`; the sort `NatWithEmpty` for a supersort of the natural numbers with an extra element `empty`; and `System` for the system, that receives as arguments a multiset of natural numbers (the idle processes), a sorted list of numbers (the processes waiting to enter the critical section), a value of sort `NatWithEmpty` (the process in the critical section), and another multiset of numbers (the processes that have already entered the critical section):

```
  sort System NatSoup NatWithEmpty .    subsort Nat < NatSoup NatWithEmpty .
  op empty : -> NatWithEmpty [ctor] .   op mtSoup : -> NatSoup [ctor] .
  op _,_ : NatSoup NatSoup -> NatSoup [ctor assoc comm id: mtSoup] .
  op _[_][_]_ : NatSoup NatList NatWithEmpty NatSoup -> System [ctor] .
```

We use the rule `ticket` to introduce a new process into the list of waiting processes:

```
  rl [ticket] : (N, NS) [NL] [NWE] NS' => NS [ins-list(NL, N)] [NWE] NS' .
```

If at least one process is waiting to enter the critical section and it contains the value `empty`, then the first process in the list is introduced into the critical section:

```
  rl [cs-in] : NS [N . NL] [empty] NS' => NS [NL] [N] NS' .
```

The rule `cs-out` moves the process from the critical section to the finished section:

```
  rl [cs-out] : NS [NL] [N] NS' => NS [NL] [empty] (N, NS') .
```

Finally, the rule `reset` moves the elements in the fourth component of the system to the first one to start the process again:

```
  rl [reset] : mtSoup [nil] [empty] NS => NS [nil] [empty] mtSoup .
endm)
```

3.2 Narrowing

Narrowing [30, 12, 22] is a generalization of term rewriting that allows free variables in terms and replaces pattern matching by unification in order to reduce these terms. It was first used for solving equational unification problems [28] and then generalized to deal with problems of symbolic reachability. Similarly to rewriting, where at each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation on the variables of the subject term and the rule's lefthand side are going to be considered. The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \Rightarrow r$ to rewrite t at a position p, but narrowing unifies the lefthand side l and the chosen subject term t before actually performing the rewriting step, while in rewriting this term must be an instance of l (i.e., only matching is required). Using this narrowing approach, we can obtain a substitution that, applied to an initial term that only contains variables (except for the function symbol at the top), generates the most general term that can apply the traversed rules.

We denote by $t \leadsto^{\sigma} t'$, with $\sigma = q_1; \ldots; q_n$ a sequence of labels, the succession of narrowing steps applying (in the given order) the statements $q_1; \ldots; q_n$ that leads from the initial term t (possibly with variables) to the term t', and by θ_σ the substitution used by this sequence, which results from the composition of the substitutions obtained in each narrowing step. We will overload the notation $t \leadsto^{q} t'$ by using conditions in q to illustrate narrowing steps due to conditions.

In the example above, we could start from the term `NS1 [NL] [NWE] NS2`, with `NS1` and `NS2` variables of sort `NatSoup`, `NL` a variable of sort `NatList`, and `NWE` a variable of sort `NatWithEmpty`, and apply one step of narrowing to obtain a set of four terms, each of them corresponding to the application of one of the rules for `System`. For example,

$$\texttt{NS1 [NL] [NWE] NS2} \leadsto^{\texttt{ticket}} \texttt{NS3 [ins-list(NL, N1)] [NWE] NS2}$$

where `NS1` has been replaced by `N1, NS3`, with `N1` and `NS3` fresh variables of the appropriate sorts (`Nat` and `NatSoup`, respectively), and then the rule has been applied. Similarly, applying `cs-in` with narrowing produces

$$\texttt{NS1 [NL] [NWE] NS2} \leadsto^{\texttt{cs-in}} \texttt{NS1 [NL2] [N2] NS2}$$

where `NWE` has been substituted by `empty` and `NL` by `N2 . NL2`, with `N2` and `NL2` fresh variables, and the rule has been applied. Similarly we could apply all the other rules (note that, due to different unifications, we can obtain several different results when applying one narrowing step to a single rule).

The latest version of Maude includes an implementation of narrowing for free, C, AC, or ACU theories in Full Maude [9]. More specifically, we are interested in the `metaNarrowSearchPath` function that, given a term and a bound on the number of narrowing steps, returns all the possible paths starting from this term, the used substitutions, and the applied rules. We use this command to generate all the reachable states in one step and then perform a breadth-first search of the state space. Note that the current implementation of narrowing only works for non-conditional rules and specifications without membership axioms; we will show in Section 5 how to check separately the conditions, including membership conditions.

4 A module transformation for narrowing

We present in this section a simple module transformation that will be applied to the modules in order to use narrowing with the equational part of Maude. This transformation has two objectives: on the one hand it transforms equations into rules (and thus it requires to transform equational conditions into rewrite conditions), which allows us to use narrowing with the equational part of Maude system modules. On the other hand, and since the current implementation of narrowing in Maude does not support memberships, we transform all the terms where membership inferences may be needed into equivalent terms with the variables declared at the kind level, while extra membership conditions stating the correct sort, that will be separately checked with the mechanisms in the next section, are added for each variable whose type has changed. More specifically, the transformation takes an equation of the form

$$l = r \text{ if } \bigwedge_{i=1}^{n} t_i = t_i' \wedge \bigwedge_{j=1}^{m} p_j := u_j \wedge \bigwedge_{k=1}^{l} v_k : s_k$$

and returns a rule

$$kind(l) \Rightarrow kind(r) \text{ if } \quad mbs(l) \wedge \bigwedge_{i=1}^{n}(kind(t_i) \Rightarrow w_i \wedge kind(t_i') \Rightarrow w_i) \wedge \bigwedge_{j=1}^{m}(kind(u_j) \Rightarrow kind(p_j) \wedge mbs(p_j)) \wedge \bigwedge_{k=1}^{l} kind(v_k) : s_k$$

where

- The terms w_i are fresh variables of the same kind as the corresponding term.
- The function *kind* replaces the sort of all the variables in the term given as argument by the corresponding kind.
- The function *mbs* generates a conjunction of conditions stating that the variables, whose type has been changed by its kind, have in fact the sort previously required, that is:

$$\begin{array}{lcl} mbs(f(t_1,\ldots,t_n)) & = & mbs(t_1) \wedge \cdots \wedge mbs(t_n) \\ mbs(c) & = & nil \\ mbs(v) & = & kind(v) : sort(v) \end{array}$$

  where f is a function symbol, the t_i are terms, c is a constant, v is a variable, and $sort(v)$ returns the sort of v.

We have to transform similarly all the membership axioms and rules in the module in order to apply them. In the membership case we obtain another membership axiom with the lefthand side and the condition transformed as shown above,[4] while rules are transformed into rules, being the equational part transformed as in the previous cases while the rewriting conditions remain unchanged.

Note that, since Maude equational modules are assumed to be confluent and terminating, the equations may be understood as oriented from left to right, which is what we are explicitly doing when transforming them into rules. Moreover, the kind transformation only postpones (but not prevents from) the checking of the specific sorts of the variables to the condition of the rule. For these reasons, it is straightforward to see that this transformation is correct, even though it can only be executed by using narrowing as explained in the next section.

If we transform the critical section example above, the membership axiom `ol2` is modified as follows (assume that the variables are now declared at the kind level):

```
cmb [ol2] : N . N' . L : SortedList
 if N : Nat /\ N' : Nat /\ L : NatList /\
    N <= N' => B /\ true => B /\
    N' . L : SortedList .
```

The first three conditions indicate that the variables, that are now declared at the kind level, have in fact the appropriate sort. The next two conditions deal with the first condition of the original axiom, `N <= N'`, which is an abbreviation for `N <= N' = true`; in this case both sides of the equality must be rewritten to the same variable `B`, defined in the kind of `Bool`. Finally, the membership condition remains unchanged. In the same way, the equation `il2` is transformed into the following rule:

```
crl [il2] : ins-list(N . SL, N') => N' . N . SL
 if N : Nat /\ SL : SortedList /\ N':Nat /\
    N' <= N => B /\ true => B .
```

while the rule `reset` becomes:

```
crl [reset] : mtSoup [nil] [empty] NS
           => NS [nil] [empty] mtSoup
 if NS : NatSoup .
```

[4] Note that this transformation generates an invalid membership axiom, because it contains rewrite conditions. However, in practice all the equations and rules in the module are unconditional and the membership axioms have been removed in order to use narrowing; these conditions and membership axioms are kept apart and checked separately by using the techniques described in Section 5.

5 Narrowing of conditional rules

We present in this section a methodology to take into account the conditions in the narrowing process because, as explained in the introduction, they are not supported by the current implementation of the Maude system. Note that other systems deal with rewrite conditions (see e.g. [22]) with a similar approach to ours: they must be solved before applying the body of the rule. The novelty of our technique, beyond describing and implementing this narrowing of conditional rules in Maude, lies on the resolution of membership conditions by means of unification.

Basically, when a rule is applied the conditions must be evaluated separately by using narrowing (remember that equational conditions become rewrite conditions) to find a substitution (that must be the composition of the substitutions obtained for each single condition) that fulfills them. If the set of conditions fulfilling the conditions is nonempty, all of them extend the set of substitutions obtained for the unconditional rule; otherwise, the rule cannot be applied.

However, in addition to rewrite conditions we must also take into account membership conditions. The current implementation of narrowing does not support membership axioms, and thus we must independently check whether a membership condition holds. The first step to achieve it was presented in the previous section: we transform the lefthand of the statements to deal with kinds instead of sorts in order to move the membership information to the conditions. The next step consists of proving the memberships (those introduced by the transformation, as well as those stated by the user); if the sort is defined by using membership axioms (and possibly by operators), then we unify the current term with the lefthand side of each membership axiom inferring this sort or any of its subsorts and then we proceed to prove the conditions in the corresponding axioms as explained before, applying the substitution obtained in the unification (moreover, it also updates the type of the variables, if they are at the kind level, to the required sorts in order to use the operator definitions, see the example below for details). Otherwise (the sort is not defined by using memberships) we update the type of the variables and the rest of the condition is processed.

In our example, we can apply conditional narrowing to the term `ins-list(NL, N1)`. The narrowing process would start by unifying this term with the lefthand side of `il2`,[5] whose transformed version was presented at the end of the previous section:

$$\texttt{ins-list(NL, N1)} \leadsto^{\textit{unif-lhs}(\texttt{il2})} \texttt{ins-list(N2 . SL1, N1)}$$

This first step requires the initial list of natural numbers `NL` to be of the form `N2 . SL1`, being `N2` and `SL1` fresh variables at the kind level. Thus, the unification generates the substitution $\texttt{NL} \mapsto \texttt{N2 . SL1}$. However, it must be extended by using the conditions of the applied rule. The first condition, `N : Nat`, is a membership condition for a sort that is not defined with membership axioms, and thus it forces the variable `N2`[6] to have sort `Nat`; we change the sort of the variable and proceed with the next condition. The second condition, `SL : SortedList`, is trickier because this sort is defined by means of membership axioms. We must use a transformed version of the membership axiom `ol2` to obtain

$$\texttt{ins-list(N2 . SL1, N1)} \leadsto^{\textit{unif-lhs}(\texttt{ol2})} \texttt{ins-list(N2 . N3 . NL2, N1)}$$

where the unification of the term with the lefthand side of the membership axiom gives the substitution $\texttt{SL1} \mapsto \texttt{N3 . NL2}$. Note that the transformation should generate three initial conditions (`N : Nat /\ N' : Nat /\ L : NatList`) that just update the sorts of the variables, two rewrite conditions, `N <= N' => B` and `true => B`, which require narrowing again to be solved, and keeps the membership condition unmodified. As we will explain in Section 8, our implementation supports some predefined operators as <=, that returns `true` when its first argument is `0`, we can use narrowing to solve the rewrite conditions:

$$\texttt{ins-list(N2 . N3 . NL2, N1)} \leadsto^{\texttt{N <= N' => B}} \texttt{ins-list(0 . N3 . NL2, N1)}$$

and the current substitution is extended with $\texttt{N2} \mapsto \texttt{0}$; $\texttt{B} \mapsto \texttt{true}$. With this substitution the next condition of `ol2` (`B => true`) trivially holds and only the membership condition, `N' . L : SortedList`, remains. It can be satisfied by using the membership axiom `ol1`, which extends the substitution with $\texttt{NL2} \mapsto \texttt{nil}$. Summarizing the narrowing process thus far, starting from the term `ins-list(NL, N1)` and applying the rule `il2` and its two first conditions (which includes applying the membership axiom `ol2` and all its conditions, the rule for <=, and the membership axiom `ol1`), we have reached `ins-list(0 . N3 . nil, N1)` with the (composed) substitution $\texttt{NL} \mapsto \texttt{0 . N3 . nil}$. We proceed now with the third condition of `il2`, `N' : Nat`, that simply updates the sort of `N1`. The next condition, `N' <= N => B`, is solved as explained above by using the substitution $\texttt{N1} \mapsto \texttt{0}$; $\texttt{B} \mapsto \texttt{true}$:

[5] Note that other rules, such as `il1` or `il3`, could be also used. In the same way, some other steps in this example could apply different membership axioms and rules.

[6] Note that, after the unification, the rule is being symbolically applied by using the substitution $\texttt{N} \mapsto \texttt{N2}$; $\texttt{SL} \mapsto \texttt{SL1}$; $\texttt{N'} \mapsto \texttt{N1}$. In the following, we will not show the substitution required to apply each rule.

$$\texttt{ins-list(0 . N3 . nil, N1)} \leadsto^{\texttt{N1 <= 0 => B}} \texttt{ins-list(0 . N3 . nil, 0)}$$

Finally, the last condition for `il2` holds because `true` is rewritten to `true`, and the rule is applied by using the obtained substitution in the righthand side, obtaining the following complete narrowing step with the substitution $\texttt{NL} \mapsto \texttt{0 . N3 . nil}$; $\texttt{N1} \mapsto \texttt{0}$:

$$\texttt{ins-list(NL, N1)} \leadsto^{\texttt{il2}} \texttt{0 . 0 . N3 . nil}$$

6 Using narrowing to generate test cases

Different testing techniques can be used to test Maude specifications, and for each of these techniques a different narrowing strategy will be used. We show in this section how to compute a coverage, how to check whether a specification fulfills an invariant, and how to examine if, given a correct specification, another module performs the required actions, which is called *conformance testing*.

6.1 Coverage criteria

Code coverage techniques [18, 24] consist of selecting a set of test cases that, when executed, apply all the statements required by the coverage criterion. In our case we use *global branch coverage* [13], a strategy that tries to find test cases that use all the statements potentially used by the function under test (which, of course, includes the functions and membership axioms in the conditions) and has been already described for functional modules in [26], and *system coverage*, an adaptation of modified condition decision coverage [18] that tries to obtain information by making the conditions to fail.

Narrowing can be naturally used to compute global branch coverage by starting with a term with variables and performing a breadth-first search, where after each narrowing step, that computes the set of reachable terms by applying one rule, we check that the conditions of each rule are fulfilled by using the mechanism presented in the previous section, thus removing some of the obtained terms and extending the substitutions when required (e.g., in the example of the previous section, the substitution was extended to $\texttt{NL} \mapsto \texttt{0 . N3 . nil}$; $\texttt{N1} \mapsto \texttt{0}$). This search finishes when all the statements required by the coverage have been used, a bound in the number of steps has been reached, or all the possible states have been reached (this last point is checked by trying to unify the terms obtained in each step with any of the previous terms; that is, we build a graph instead of a tree). Moreover, our system provides two different options to select the set of test cases: the smallest one, composed of the minimum number of terms whose execution leads to the execution of all the statements in the coverage and thus may contain complex test cases; and the simplest one, in the sense that it may present more but simpler test cases. The user can switch between these two modes to decide which one is more appropriate for each specification.

More formally, we look for a set of sequences σ_i and terms t_i, $0 \leq i \leq l$, such that, given the set of labels Q defining the coverage and a term $f(v_1, \ldots, v_n)$, with f the function under test and v_i variables of the appropriate sorts, $\forall q \in Q \; \exists_{i=0}^{l} \; . \; f(v_1, \ldots, v_n) \leadsto^{\sigma_i} t_i \; \wedge \; q \in \sigma_i$. The test cases will be $\bigcup_{i=0}^{l} \theta_{\sigma_i}(f(v_1, \ldots, v_n))$. Since there are several different possibilities to select the σ_i, the different strategies to display the set of test cases will choose between a small number of large sequences, that will generate less test cases applying more rules, and a big number of short sequences, that will generate simpler test cases. Note that the extension to testing of system modules is straightforward; in this case we want to test the transitions of the terms with a given sort instead of a specific function, and thus the narrowing process starts with a term with variables of the given sort, and tries to apply all the reachable rules, equations, and memberships, proceeding in the same way as the testing for functional modules. That is, we start the narrowing process from $c(v_1, \ldots, v_n)$, with c any constructor for the sort and v_i variables of the appropriate sorts, and continue as indicated above.

In our lists example, we may be interested in testing the function `ins-sort` using the global branch coverage criterion. This function is defined by two equations (`is1` and `is2`); one of these equations uses the function `ins-list`, and thus its three equations (`il1`, `il2`, and `il3`) must also be added to the needed coverage; finally, this function uses the functions `_<_` and `_<=_`, imported from `NAT` (more details about these functions are provided in Section 8) and a variable of sort `SortedList`, which is defined with two membership axioms (`ol1` and `ol2`). All these statements must be executed at least once by the test cases to fulfill global branch coverage. We can use our tool to automatically generate the test cases, following the default strategy that selects the smaller set of test cases:

```
Maude> (test ins-sort .)
 1 test cases have to be checked by the user:
  1. ins-sort(1 . 0 . 0 . 0 . nil) has been reduced to 0 . 0 . 0 . 1 . nil
All the statements were covered.
```

```
ins-sort(L) ⇝is2
ins-list(ins-sort(L1), N1) ⇝is2
ins-list(ins-list(ins-sort(L2), N2), N1) ⇝is2
ins-list(ins-list(ins-list(ins-sort(L3), N3), N2), N1) ⇝is2
ins-list(ins-list(ins-list(ins-list(ins-sort(L4), N4), N3), N2), N1) ⇝is1
ins-list(ins-list(ins-list(ins-list(nil, N4), N3), N2), N1) ⇝il1
ins-list(ins-list(ins-list(N4 . nil, N3), N2), N1) ⇝il3
ins-list(ins-list(0 . ins-list(nil, s(N5)), N2), N1) ⇝il1
ins-list(ins-list(0 . s(N5) . nil, N2), N1) ⇝il2,ol1
ins-list(0 . 0 . s(N5) . nil, N1) ⇝il2,ol2,ol1
0 . 0 . 0 . s(N5) . nil
```

Figure 2: Narrowing path for global branch coverage

Note that the tool shows the initial term, the result of reducing it in the module, and whether some reachable statements could not be used. The term shown by the tool may be obtained as shown in Figure 2, where `s` stands for the successor function over natural number (note that this is one branch of a search tree of depth 10).

Since all the possible instantiations of this term generate test cases traversing all the required statements, the tool generates the simplest one by replacing the variables with constants of the given sort (or the simplest built term if the sort does not have constants). If we find that any reduction is wrong, we could debug it with:

```
Maude> (invoke debugger with user test case 1 .)
Declarative debugging of wrong answers started.
...
```

This command starts the declarative debugging process [27] that, by asking questions to the user about the computations that took place will find the specific statement that generated the wrong behavior. This command is available for all the testing options.

Moreover, we propose another coverage criterion, related to modified condition decision coverage (MCDC) [18, 10]. Basically, MCDC requires that all the conditions in a program are evaluated with the given set of test cases to both true and false. In a non-deterministic framework as the one of system modules it is important to know, as explained in [27], the applied statements that make the program reach certain states, the *positive information*, but also the statements that were not applied and thus prevented the program from reaching some other values, the *negative information*. While we obtain the positive information with the global branch coverage shown above, it does not provide any of the negative information. For this reason, we have studied the different possibilities arising in Maude system modules to obtain negative information, to keep track of the rules that could not be applied:

- It should not consider as negative information trivial failures, which happen in general when we try to unify terms built with different operators.
- However, asking the term to match the lefthand side may in general be too restrictive, because it can contain sort information significant for the testing process. However, this cannot happen in our framework because we have changed the type of the variables in the lefthand sides by the corresponding kinds.
- Finally, we must now decide whether it is necessary to force all the conditions in the rules to fail or not. However, this requirement would be too difficult to satisfy because most conditions are used in general as syntactic sugar (e.g. matching conditions may be used for fixing large values that are later used in several places of the same equation), and hence we determined that the coverage should only require, for each rule, the failure of one of its atomic conditions.

For these reasons, we have also implemented the so-called system coverage criterion, which requires a set of test cases to apply all the rules in the transformed module (which corresponds to global branch coverage) but also to fail for at least one condition for each rule in the *original* module. Note that this coverage criterion is specially well suited for specifications with conditional rules or with membership information on the lefthand side of the rules; since our example does not have this kind of rules, it is not possible to find terms providing negative information and thus it is not worth applying this coverage, that is activated with the command:

```
Maude> (system coverage .)
System Coverage selected
```

6.2 Checking invariants

Checking of invariants has already been studied for Maude specifications in [8, Chapter 12]. It takes advantage of the command `search`, that performs a breadth-first search from an initial term, given a bound in the number of steps and a condition to be fulfilled; by using the *negation* of the condition to be fulfilled we can check that no illegal states are reached. We apply a similar idea in our testing framework by using symbolic search; this search will traverse all the possible states and, each time a rule of the *original* program is applied, it tries to find a path to fulfill the negation of the invariant. If such a path is found, then the specification does not fulfill the invariant. Note that the invariant is usually specified by using equations, and thus it is important to use equations in the narrowing process, since it allows the tool to fix the values of the initial state required to fulfill (the negation of) the condition.

More formally, we consider a new rule $inv(pat) \Rightarrow pat\ if\ Cond$, where *inv* is a new operator defined over the sort of states, *pat* is the pattern given for the invariant, and *Cond* a condition (we assume the invariant is composed of a pattern and a condition, see below for details). Thus, for every narrowing sequence $t \leadsto^{q_1} t_1 \leadsto^{q_2} \ldots \leadsto^{q_n} t_n$, the invariant is fulfilled if, for every t_i obtained by using a narrowing step with the rule q_i we cannot find a term t' such that $t_i \leadsto^{inv} t'$ (we look for the negation of the invariant). If such a term exists, then the term $\theta_{inv}(\theta_{q_1;\ldots;q_i}(t))$ can be used as initial term for debugging; otherwise, the invariant holds.

The transformation presented in Section 4 allows us to check invariants in both functional and system modules. We could e.g. set an invariant on our critical section example stating some correct property over lists or systems, but it is worth examining how an initial term proving the specification wrong is obtained. In our critical section example we can specify a function `empty?`, that checks whether a `NatSoup` is empty, defined by equations as follows:

```
op empty? : NatSoup -> Bool .
eq [mt1] : empty?(mtSoup) = true .
eq [mt2] : empty?(NS) = false [owise] .
```

and then search for a system that never has its first argument `empty` (remember that we look for the negation of the invariant) with:

```
Maude> (test [10] System =>+ NS1:NatSoup [NL:NatList] [NWE:NatWithEmpty]
                             NS2:NatSoup s.t. empty?(NS1:NatSoup) .)
1 test cases have been checked:
  1. The term mtSoup [nil] [0] 0 reaches the state
     mtSoup [nil] [empty] (0,0), which does not fulfill the invariant.
```

This command looks for terms of sort `System` that, in at least one step (indicated by the search arrow `=>+`; the tool also provides searches in zero or more steps with `=>*` and searches for final forms with `=>!`) and at most `10`, match the pattern and fulfill the condition (the negation of the invariant). In this case, the tool has found (as expected) an initial state that, after applying one rule (in this specific case it is `cs-out` although it would be possible to apply other rules), reaches a state that does not fulfill the invariant. In this case the narrowing process has fixed the value of the first `NatSoup` to `mtSoup` to fulfill the condition and has forced an element to be in the critical section to apply the rule, while the `nil` list and the singleton soup are just possible instances of the variables left by the narrowing process:[7]

$$\texttt{NS1 [NL] [NWE] NS2} \leadsto^{\texttt{cs-out}} \texttt{NS1 [NL] [empty] (N1, NS2)}$$

and then checking the property by instantiating `NS1` with `mtSoup` when applying the equations for `empty?`:

$$\texttt{empty?(NS1)} \leadsto^{\texttt{mt1}} \texttt{true}$$

6.3 Conformance testing

Conformance testing [31, 14, 5] involves testing a system with respect to its specification. The goal of this approach is to check that the system presents the same behavior as the specification, that has already been tested. To check whether an implementation conforms to a specification we must formalize the conformance notion by means of an *implementation relation* that relates the two systems. In our case, and taking into account that a rewrite system can be understood as a labeled transition system, where terms stand for states and rewrites for transitions, we apply to Maude specifications the conformance testing strategies for such systems [31]. In particular, we use the relation **conf** [4], that requires the implementation to perform the same actions as the specification, although it allows

[7] Note that in this case the tool selects as simplest `NatSoup` the singleton one containing `0` instead of `mtSoup`, that may seem simpler. This happens because both of them are constant values and `0` corresponds to a smaller set.

the implementation to execute some other actions not included in the specification, that is, **conf** requires that an implementation does what it should do, not that it does not do what it is not allowed to do.

In our framework we consider that only the rules in the original specification must be executed in the implementation, and thus narrowing steps using equations are considered auxiliary and is not required to reproduce them in the implementation. In this way, we compute all the possible paths by using narrowing in the specification and then that all these paths are also possible in the implementation. More formally, if we denote by $\sigma|_R$ the restriction of σ to the rules in R, that is, remove from σ all those statements that are not in the set, then we require that for every narrowing sequence $t \rightsquigarrow^{\sigma_s} t_s$ in the specification there exists a narrowing sequence $t \rightsquigarrow^{\sigma_i} t_i$ in the implementation such that $\sigma_s|_R$ is a prefix of $\sigma_i|_R$. Note that, although the reached states may be different in the specification and the implementation (only the applied rules matter), we consider that both the correct specification and the system being tested share the same signature for the initial terms and the same rule labels; this can be achieved by means of a renaming.

For the sake of example, we could create a new module `RED-CS` that has the same rules as `CS` except for the rule `cs-out`. We can state `CS` as the specification with:

```
Maude> (correct test module CS .)
CS selected as correct module for testing.
```

Now, we can check the behavior of `RED-CS` with respect to this module with:

```
Maude> (test in RED-CS : System .)
  Starting from the term 0 [nil] [0] 0 the rule
  cs-out could not be applied to the implementation.
```

That is, the tool shows the simplest term (in fact, only the `0` in the critical section is instantiated during the process) that is required to find the disconformity between the specification and the implementation due to the `cs-out` rule.

7 Trusting

Our tool provides some trusting techniques to enhance its performance. Basically, it only takes into account labeled statements when computing coverages and checking the implementation relation. Moreover, the user can also select a subset of these statements by using the different commands available in the tool (trusting of all the statements of a given module, trusting of a complete kind of statements—e.g. all the equations, memberships, or rules—and trusting of single statements). Using these commands we can use different trusting strategies: assuming that our specifications are structured, we can test first easier specifications, and then trust them when testing larger specifications including them; and we can trust all the equations (except for the ones defining the property when checking invariants) and memberships when testing system modules. Of course, trusting mechanisms are correct assuming the user points out as trusted only rules that are not relevant for the testing process; see the example below to see the risks of reckless trusting.

Trusting works in a different way depending on the testing strategy: if we are computing a coverage then the trusted statements are removed from the needed coverage, and thus we may reduce both the test cases required for testing the specification and the depth of the narrowing search. When using conformance testing, the trusted statements are related to the specification and indicate that the behavior specified by the rule is not required to be performed in the system being tested (e.g. because it is an auxiliary rule). That is, the sequences of statements σ required for coverage are not required to contain the trusted statements,[8] while the restriction to rules in the specification given in conformance testing is now applied to non-trusted rules in the specification.

For example, we can trust the statements `il3` and `ol2` (which required the longest computations in Figure 2) if we are sure of its correctness to improve the performance of the computation of the global branch coverage in Section 6.1 by using the commands:

```
Maude> (test include SORTED-NAT-LIST .)
  Labels hd il1 il2 il3 is1 is2 ol1 ol2 have been added to the coverage.
Maude> (test deselect il3 ol2 .)
  Labels il3 ol2 have been excluded from the coverage.
Maude> (test in SORTED-NAT-LIST : ins-sort .)
  1 test cases have to be checked by the user:
     1. ins-sort(0 . 0 . 0 . nil) has been reduced to 0 . 0 . 0 . nil
  All the statements were covered.
```

[8] Note that using trusting when using system coverage will remove the statements from both the positive and negative information.

```
ins-sort(L) ⇝is2
ins-list(ins-sort(L1), N1) ⇝is2
ins-list(ins-list(ins-sort(L2), N2), N1) ⇝is2
ins-list(ins-list(ins-list(ins-sort(L3), N3), N2), N1) ⇝is1
ins-list(ins-list(ins-list(nil, N3), N2), N1) ⇝il1
ins-list(ins-list(N3 . nil, N2), N1) ⇝il2
ins-list(0 . N3 . nil, N1) ⇝il2,is1
0 . 0 . N3 . nil
```

Figure 3: Narrowing path for global branch coverage with trusting

Obtaining in this case a simpler test case that covers all the statements. It is interesting to see that trusting a rule when using conformance provides more flexibility, because it allows to perform some analysis by removing auxiliary rules that are not supposed to be applied in the final implementation. However, if the user trusts a statement that should not be trusted it may obtain an incorrect answer, hence the assumption presented above about the correctness of trusting, that may produce incorrect results.

Similarly, we can trust the rule `cs-out` when using conformance testing and check that in this case the specification and the implementation perform the same actions:

```
Maude> (test include CS .)
  Labels cs-in cs-out hd il1 il2 il3 is1 is2 ol1
  ol2 reset ticket have been added to the coverage.
Maude> (test deselect cs-out .)
  Labels cs-out have been excluded from the coverage.
Maude> (test in RED-CS : System .)
  The implementation conforms to the specification.
```

Notice that trusting the rule `cs-out` makes the implementation conform to the specification. This means that trusting not only improves the performance of the tool, it also allows to perform some analysis by removing auxiliary rules that are not supposed to be applied in the final implementation. However, if the user trusts a statement that should not be trusted it may obtain an incorrect answer, hence the assumption presented above about the correctness of trusting, that may produce incorrect results.

The improvement in the performance when using trusting is highly dependent on the selected set of statements: while in some cases trusting may reduce the number of steps more than a 50%, in other cases they are not reduced at all. For example, the global branch coverage obtained in Section 6.1 was highly reduced by trusting the statements shown above, reducing the depth of the search tree from 10 to 7, as illustrated in Figure 3.[9] However, selecting other statements such as `is1` or `is2`, that must be always executed in order to reach a state where other statements can be used, would not reduce the size of the search at all.

All the examples in this paper, and much more information (including examples for conformance testing) is available at `http://maude.sip.ucm.es/testing/`.

8 Implementation

As outlined in the introduction, the current implementation of the system has been developed in Maude itself by taking advantage of the reflective capabilities of rewriting logic [21], that allow the programmer to manipulate modules and statements as usual data. In this way, we can check whether the module being tested has a correct theory (free, A, AC, or ACU); transform the module by accessing the different equations, membership axioms, and rules and transforming them; and use these new rules to perform the breadth-first narrowing used to compute the test cases presented in the previous sections. Furthermore, our tool extends Full Maude [8, Chap. 18], that enhances the input/output interactions with the user with features for parsing, evaluating, and pretty-printing terms.

For example, the `eq2rl` function below receives a module and an equation (where the variable `AtS` stands for a set of attributes and the rest of variables are self-explaining) and returns a pair with (i) an unconditional rule corresponding to the transformed equation with the sort of the variables changed by its kind with the function `up2kind` and (ii) a partial function relating the label of the equation with the membership axioms generated for the variables, that is computed with the function `mbs`. A similar equation is used to transform conditional equations.

[9] Remember that this is one branch of the search tree, that is, trusting has reduced the depth of the search tree from 10 to 7, which results in a huge improvement of the performance.

```
op eq2rl : Module Equation -> RSPair .
eq eq2rl(M, eq T = T' [AtS] .) =
 < rl up2kind(M,T) => up2kind(M,T') [AtS] ., label(AtS) |-> mbs(M,T) > .
```

Regarding the application of narrowing, we use the function `metaNarrowSearchPath` available in Full Maude which, as introduced in Section 3.2, returns a set of traces indicating, for each step of the trace, the reached term, its type, the applied substitution, and the label of the rule that has been used. Using this command for traces with only one step we obtain all the reachable states from a given term; for each one of the terms in the set we use the map obtained when transforming the module to check the conditions. If all the conditions hold by using the same narrowing strategy, we obtain a set of substitutions that must extend the substitution initially generated; each substitution thus obtained must be applied to the term to continue with the process. Note that the negative information for each step can be easily obtained: the rules that have been executed in the initial narrowing step but have been discarded when the conditions have been checked are the ones providing this kind of information.

Moreover, the test case generator deals with some predefined functions: we keep the `owise` attribute, that indicates that an equation can only be applied when all the other equations for this function cannot, as metadata in the transformed rule and it is only used when narrowing fails to apply the others equations for the given operator symbol or the substitution cannot be used on them; some built-in Boolean functions, like `if_then_else_fi`, have been defined by means of rules;[10] and most of the built-in arithmetic functions, like the `_<_` and `_<=_` in the example in Section 3.1, have also been defined with rules. Note that all these extra rules are unlabeled and thus they are trusted.

The current implementation of the system outperforms by far the implementation of the test-case generator for functional modules in [26] that, as explained in the introduction, generated terms by using all the possible combinations of constructors and then executed each of these terms to check the statements used during the computation. This blind generation wasted most of the time checking terms that were equivalent from the point of view of testing, and thus it could not generate test cases covering all the statements of small specifications such as balanced trees, which require trees of several levels as input to apply all the equations. The new implementation generates test cases much faster and covers more statements thanks to the narrowing process, that directs the use of the constructors.

9 Concluding remarks and ongoing work

We have presented in this paper how to use narrowing to generate test cases for Maude specifications. To achieve this we use a module transformation that allows us to use the equational part of Maude modules in the narrowing process and a method to check whether the conditions of the applied statements are fulfilled, including those conditions that require membership axioms. Using these techniques we have implemented a tool that is able to compute a set of test cases fulfilling two different coverage criteria, to check whether an invariant is fulfilled by the specification, and to examine whether an implementation of the system fulfills the behavior indicated by its specification. Moreover, two different sets of test cases can be computed: a smaller set that contains more complex terms or a larger set that contains less complex terms; the user is in charge of selecting the most appropriate depending on the complexity of the specification and his knowledge of it. Trusting mechanisms are also provided to improve the performance of both coverage criteria and conformance testing. Finally, some predefined modules can be also used to generate the test cases.

As future work, we plan to extend the tool to introduce symbolic model checking [11], that would allow the user to check linear temporal logic formulas over the specification starting from a term with variables, thus proving the formula on, potentially, all the possible inputs of the system. Moreover, we are studying new coverage criteria and implementation relations to allow the user to choose the most appropriate technique for each application. Finally, we also intend to develop a distributed implementation of the tool to deal with narrowing; in this way, we can start the symbolic search of the system in one Maude instance and then send the different paths to different Maude processes, that must share some information (the coverage and the reached states) to finish the search as soon as possible.

References

[1] B. Beizer. *Software testing techniques*. Dreamtech, 2002.

[10] Note that this function is *universal*, that is, the second and third arguments may take values of any sort; for this reason we declare different rules for all the kinds in the specifications.

[2] P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook, editors. *Testing Techniques in Software Engineering. Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*, volume 6153 of *Lecture Notes in Computer Science*. Springer, 2007.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[4] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. *Protocol Specification, Testing, and Verification*, VI:349–360, 1987.

[5] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC 2007*, pages 1292–1297. IEEE, 2007.

[6] J. Christiansen and S. Fischer. Easycheck – test data for free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming, FLOPS 2008*, volume 4989 of *Lecture Notes in Computer Science*. Springer, 2008.

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.6)*, January 2011. `http://maude.cs.uiuc.edu/maude2-manual`.

[10] A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the 19th Digital Avionics Systems Conference, DASC 2000*, volume 1, pages 1B6.1–1B6.7, 2000.

[11] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Proceedings of the 18th International Conference on Term Rewriting and Applications, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.

[12] M. Fay. First-order unification in an equational theory. In W. H. Joyner, editor, *Proceedings of the 4th Workshop on Automated deduction*, Academic Press, pages 161–167, 1979.

[13] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP 2007*, pages 63–74. ACM Press, 2007.

[14] M.-C. Gaudel. Software testing based on formal specification. In P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcock, editors, *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE 2007, Revised Lectures*, volume 6153 of *Lecture Notes in Computer Science*, pages 215–242. Springer, 2010.

[15] M. Gomez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming*, 10:659–674, 2010.

[16] R. M. Hierons, K. Bogdanov, J. P. Bowen, J. D. Rance Cleaveland, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[17] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of the 7th international conference on Logic for programming and automated reasoning, LPAR 2000*, volume 6397 of *Lecture Notes in Computer Science*, pages 131–160. Springer, 2000.

[18] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 95–107. ACM, 1994.

[19] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.

[20] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1, 1992.

[21] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 1992.

[23] R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic Java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.

[24] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14:868–874, June 1988.

[25] C. Pacheco. *Directed Random Testing*. PhD thesis, Massachusetts Institute of Technology, June 2009.

[26] A. Riesco. Test-case generation for Maude functional modules. In *Proceedings of the 20th International Workshop on Algebraic Development Techniques, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[27] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.

[28] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[29] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.

[30] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[31] J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.